Multi-start local search algorithms on GPU

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi

Multi-start local search algorithms on GPU

Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi

INRIA Dolphin Project / Opac LIFL CNRS
40 avenue Halley, 59650 Villeneuve d'Ascq Cedex FRANCE.
The-Van.Luong@inria.fr, [Nouredine.Melab, El-Ghazali.Talbi]@lifl.fr

1 Introduction

In practice, combinatorial optimization problems are complex and computationally time-intensive. Even if local search (LS) algorithms allow to significantly reduce the computation time cost of the solution exploration space, the use of parallelism is required to accelerate the search process. Indeed, LSs are inherently parallel and three parallel models [1] are often used to solve efficiently large combinatorial problems: algorithmic-level (multi-start model), iteration-level (parallel evaluation of the neighborhood), and the solution-level (parallel evaluation of a single solution).

Nowadays, GPU computing has been widely used to solve challenging problems in science and engineering. One of the major issues of that emerging technology is the design of algorithms that allow to efficiently use the huge amount of resources at disposal. To take benefit from the large amount of resources provided by the GPU, the three parallel models must be adapted to take into account the characteristics of the environment: distribution of data processing between the CPU and the GPU, the thread synchronization, the capacity constraints of these memories, etc.

The major objective of our work is to re-visit the parallel models for metaheuristics in order to take into account the above characteristics. In [2], we have proposed to re-design the iteration-level parallel model for LSs on GPU. To go on this way, the main objective of this paper is to deal with the algorithmic-level on GPU architectures where many LSs are executed in parallel. More exactly, we propose to study different schemes of deployment for the design of multi-start LSs on GPU based on popular hill climbing (HC), simulated annealing (SA) and tabu search (TS).

2 Algorithmic-level model on GPU

Despite the fact that the algorithmic-level model has already been applied in some previous works in the context of the TS on GPU [3], it has never been widely investigated in terms of memory management. Therefore, we propose in this paper to focus on finding efficient associations between the different available memories and the data commonly used in the multi-start LS algorithms.

A natural way for designing multi-start LSs on GPU is to parallelize the whole LS process on GPU by associating one GPU thread with one LS. This way, the main advantage of this approach is to minimize the data transfers between the host CPU memory and the GPU. Figure 1 illustrates this idea of full distribution.

Regarding the data management on GPU, similar observations can be made whatever the used multi-start LS algorithm:

- **Global memory:** For each running LS on GPU (one thread), its associated solution and resulting fitness are stored on the global memory. In a general way, all the data in combinatorial problems could be associated with this memory. However, since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories.
- **Texture memory:** This read-only memory is adapted to LS algorithms since the problem inputs do not change during the execution of the algorithm. In most of optimization problems, problem inputs do not often require a large amount of allocated space memory. As a consequence, these structures can take advantage of the 8KB cache per multiprocessor of texture units. Moreover, cached texture data is laid out to give best performance for structures with 1D/2D access patterns such as matrices.
- **Constant memory:** This memory is read only from kernels and is hardware optimized for the case where all threads read the same location. It might be used when the calculation of the evaluation function requires a lookup table.

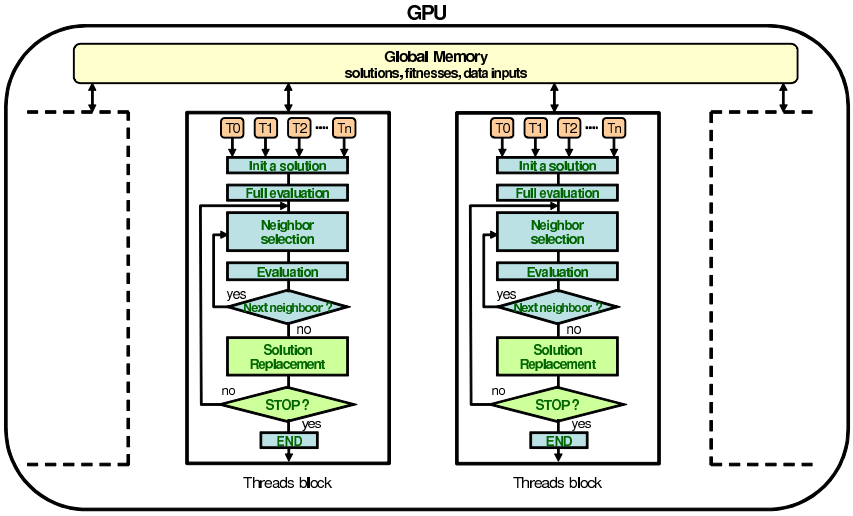

Fig. 1. Full distribution of local searches on GPU. One thread is associated with one local search.

- **Shared memory:** This fast memory is located on the multiprocessors and shared by threads of each thread block. It might be used in the case of cooperative multi-start LSs to share the best-so-far solution.
- **Registers:** Among streaming processors, they are partitioned among the threads running on it and they constitute fast access memory. In the kernel code, each declared variable is automatically put into registers.
- **Local memory:** In a general way, additional structures such as declared array will reside in local memory. In fact, local memory resides in the global memory allocated by the compiler and its visibility is local to a thread (a LS).

For the management of random numbers in SA, efficient techniques are provided in many books such as [4] to implement random generators on GPU. For deterministic multi-start LSs based on HC or TS, the random initialization of solutions might be done on CPU and then they can be copied on the GPU via the global memory to perform the LS process. This way, it ensures that the obtained results are the same as a multi-start LS performed on a traditional CPU.

Regarding the management of the tabu list on GPU, since the list is particular to a TS execution, a natural mapping is to associate a tabu list to the local memory. However, since this memory has a limited size, large tabu lists should be associated with the global memory.

3 Experiments

To validate our approach, the algorithmic-level has been implemented on the quadratic assignment problem (QAP) on GPU using CUDA. The used configuration is an Intel Xeon 3GHz with a GTX 280 (32 multiprocessors). The obtained results are reported in Table 1 for each multi-start algorithm.

Since the number of threads per block is arbitrary fixed to 256 and the number of registers per kernel varies according to the instance, the multiprocessor occupancy is not optimal. That is the reason why, the reported speed-ups are quiet irregular. Nevertheless, the point to highlight is that the texture optimization (GPUTex) on data inputs allows to improve the speed-ups in comparison with a standard GPU version where inputs are stored in the global memory (GPU).

Another experiment consists in measuring the speed-ups by varying the number of TSs for the instance tai50a (Fig 2). Moreover, we propose to compare this approach with the iteration-level model [2] (parallel evaluation of neighborhood). As we can see, the algorithmic-level starts providing some acceleration from a number of 512 threads. Regarding the iteration-level, the obtained speed-ups are quiet regular whatever the number of LSs. Finally, it becomes more interesting to use the algorithmic-level from a number of 2048 TSs.

Table 1. Measures of the efficiency of the algorithmic-level on the QAP. The average time is reported in seconds for 30 executions, the number of LSs is fixed to 4096.

	tai30a	tai35a	tai40a	tai50a	tai60a	tai80a	tai100a
HC CPU	5.48	10.71	17.18	44.56	88.32	302.43	810.39
HC GPU	$3.19_{\times 1.7}$	$4.99_{\times 2.1}$	$7.44_{\times 2.3}$	$15.79_{\times 2.8}$	$30.06_{\times 2.9}$	$90.45_{\times 3.3}$	$224.51_{\times 3.6}$
HC GPUTex	$0.75_{\times 7.3}$	$1.38_{\times 7.8}$	$2.43_{\times 7.1}$	$6.69_{\times 6.7}$	$15.34_{\times 5.8}$	$56.28_{\times 5.4}$	$143.78_{\times 5.6}$
SA CPU	412.64	636.13	874.44	1672.63	2699.89	6807.88	13960.69
SA GPU	$92.39_{\times 4.5}$	$147.04_{\times 4.3}$	$228.98_{\times 3.8}$	$428.79_{\times 3.9}$	$749.51_{\times 3.6}$	$1720.87_{\times 4.0}$	$3459.15_{\times 4.0}$
SA GPUTex	$34.52_{\times 12.0}$	$57.05_{\times 11.2}$	$90.35_{\times 9.7}$	$205.74_{\times 8.1}$	$412.91_{\times 6.5}$	$1186.13_{\times 5.7}$	$2426.11_{\times 5.8}$
TS CPU	335.57	531.35	725.39	1539.60	2439.86	6097.61	13004.76
TS GPU	$85.41_{\times 3.9}$	$152.08_{\times 3.5}$	$228.49_{\times 3.2}$	$414.50_{\times 3.7}$	$712.74_{\times 3.4}$	$1632.36_{\times 3.7}$	$3222.01_{\times 4.0}$
TS GPUTex	$31.14_{\times 10.8}$	$51.06_{\times 10.4}$	$77.71_{\times 9.3}$	$176.29_{\times 8.7}$	$355.59_{\times 6.9}$	$993.25_{\times 6.1}$	$2083.35_{\times 6.2}$

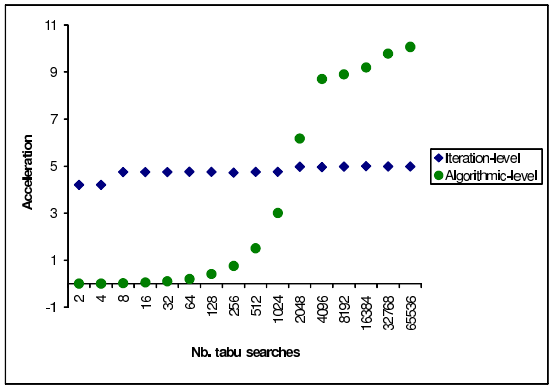

Fig. 2. Measures of the speed-up of the algorithmic-level approach in comparison with the iteration-level by varying the number of tabu searches (tai50a).

4 Conclusion

We have proposed in this paper a guideline to design and implement GPU-based multi-start LS algorithms. Indeed, only an efficient memory management allows to provide significant speed-ups (up to $\times 12$). However, to take full advantage of the algorithmic-level, it needs to be run for a large number of LSs. Therefore, for a small number of LSs, the iteration-level on GPU might be prefered.

A perspective of this work is to combine the algorithmic-level on GPU with a multi-core approach. Indeed, since this model has a high degree of parallelism, the CPU cores can also work in parallel in an independent manner. Moreover, since nowadays the actual configurations have 4 and 8 cores, instead of waiting the results back from the GPU, this computational power should be well-exploited in parallel to provide additional accelerations.

References

1. Alba, E., Talbi, E.G., Luque, G., Melab, N.: 4. Metaheuristics and Parallelism. In: Parallel Metaheuristics: A New Class of Algorithms. Wiley (2005) 79–104
2. Luong, T.V., Melab, N., Talbi, E.G.: Local search algorithms on graphics processing units. a case study: the permutation perceptron problem. In: EvoCOP. Volume 6022 of LNCS., Springer (2010) 264–275
3. Zhu, W., Curry, J., Marquez, A.: Simd tabu search with graphics hardware acceleration on the quadratic assignment problem. International Journal of Production Research (2008)
4. NVIDIA: GPU Gems 3. Chapter 37: Efficient Random Number Generation and Application Using CUDA. (2010)